# COMP 6360 Wireless and Mobile Networks
# Project 2 Report

Team 2*

April, 2012

## 1 Architecture

The simulator consists of four threads and multiple classes. As shown in Figure 1, they work collaboratively to simulate the OLSR Protocol and EEBL application. Section 2 discusses details of each part of the system.

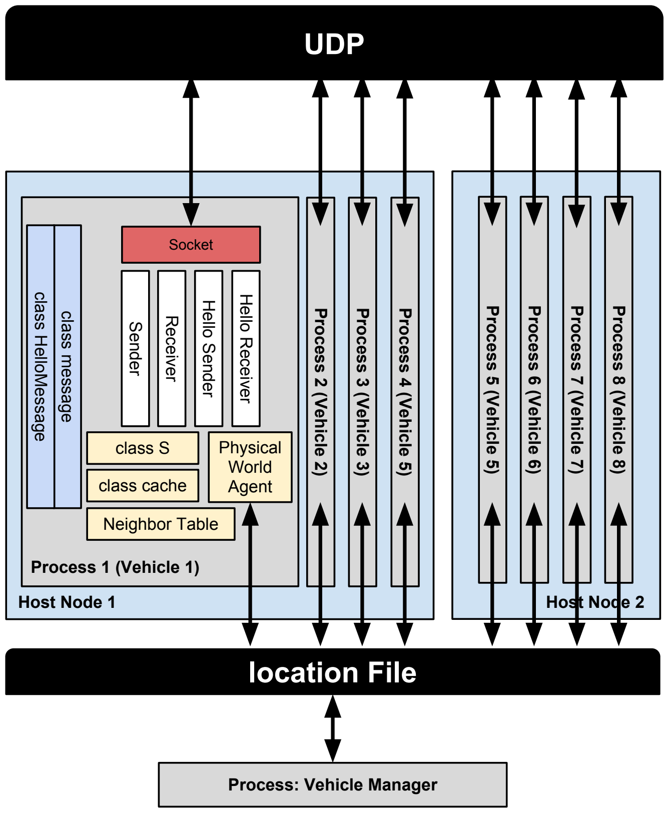


Figure 1: Architecture

## 2 Design

A lot of modules implemented in our project 1 are reused, including the message modeling and serialization, fast cache for EEBL and beacon messages, configuration class, etc.. Several new modules are designed and implemented for the sake of:

1. Using OLSR's MPR concept for rebroadcasting decisions;
2. Implementing dynamic traffic, i.e., the vehicles are mobile.

This section describes in detail design of new modules introduced in project 2.

### 2.1 OLSR building blocks

**Involved classes:** `Neighbor Table, MPR selectors Table, Hello Message`

`Neighbor Table` class is in charge of managing neighbors and computing MPRs. It has three main responsibilities:

#### 2.1.1 constructing neighbor table from Hello Messages

The Neighbor Table class constructs a table from Hello messages, which stores all the 1 hop neighbors and also neighbors of them. We use C++ std::map as the data structure to hold this table. Table 1 shows an example of neighbor table for Node 1.

| 1 hop neighbors | neighbors of it |
|---:|:---:|
| 2 | 1, 3, 4, 8 |
| 5 | 1, 4, 6, 7 |
| 8 | 1, 2 |

Table 1: An example of neighbor table

When a new Hello message is received, Neighbor Table class checks whether the originator of the message is already in neighbor table or not. If it is, neighbors of the originator and also timestamp for this entry is updated. Otherwise a new entry is added.

#### 2.1.2 Remove old entries

In addition to carrying data, `Neighbor Table` class is also in charge of removing entries older than 3s, since previous neighbors may already moved away and no more Hello messages will be received from them. In order to do this, there is a timestamp associated with every entry in the table, in every 1s the update() method of this class is invoked which will remove old entries.

*Jiao Yu [jzy0012], Song Gao [szg0031], Xinyu Que [xzq0003] (ordered by first name)

#### 2.1.3 Compute MPRs

The algorithm to select MPRs should try to cover all 2 hop neighbors while minimizing the number of MPRs. We come up with an algorithm: First select the node which covers the maximum number of 2 hop neighbors as an MPR, then subtract the 2 hop neighbors covered by it from the entire set of 2 hop neighbors, repeat this step until the remaining 2 hop set is empty. This method may not be optimal, but it serves well for our purpose.

### 2.2 MPR selectors Table

The MPR selectors Table stores data about which nodes select me (the current node) as an MPR, so the current node will know whether to rebroadcast an eebl message by looking up this table. The data structure for this table is easy, it just needs to store a vector of node IDs and also their associated timestamp. By receiving an Hello Message, the MPR selectors class check the link status indicated by this Hello message. If it is MPR, then add the originator of it to that vector if the originator was not already there, otherwise update its timestamp. This class has a similar update() method as in Neighbor Table class, which will remove entries older than 3s.

### 2.3 Hello Message

Hello Message class has three fields: Originator ID, link status, and neighbors. In order to avoid variable length of packet size, which will complicate socket transmission of Hello Messages, we give the ”neighbors” field fixed size of 120 bytes, which will hold up to 30 neighobrs ID. Usually a vehicle will have less than 30 one hop neighbors, the unused space of the 120 bytes are filled with unsigned int::MAX to indicate the end of this field.

The construction of a hello message should be based on neighbor table and the receiver node. From neighbor table we are able to extract the current node's one hop neighbors and MPRs. After putting originator ID and neighbors data into a hello message, the link status should be set properly according to whether the receiver node is one of the current node's MPRs.

### 2.4 Threads Implementation

We split the program into 4 threads,

```
void * sender_main(void * context)
void * recver_main(void * context)
void * sender_hello_main(void * context)
void * recver_hello_main(void * context)
```

, to do the job of sending eebl/beacon packets, receiving eebl/beacon packets, sending hello packets and receiving hello packets respectively. Hello messages are of different size of eebl/beacon messages, to simplify the problem, we use different socket addresses to transmit them, which are listed in configuration file. In the following subsections, more details of each thread are discussed.

#### 2.4.1 Sender Thread

The functionality of Sender Thread is basically the same as in project 1, but there are two differences: 1. It now gets the set of nodes within communication range from the Mobility module, not from configuration file any more. 2. We let Node 7 send out eebl packet every 0.1s, instead of letting every vehicle generate eebl randomly with p = 0.1 as in project 1.

#### 2.4.2 Receiver Thread

The functionality of Recver Thread is basically the same as in project 1, the only difference is the way it determines whether to rebroadcast an eebl message. Different from project 1 which is based on a probability model incorporating distance, in this project rebroadcasting is determined by whether the sender of the eebl message is in the current node's MPR selectors table or not.

#### 2.4.3 Sender Hello Thread

This thread is responsible to construct hello messages, send them to its neighbors accordingly, and also remove old entries from neighbor table and MPR selectors table. In every one second, this thread updates the two tables first, get the current nodes in communication range from Mobility module, get the current neighbors and MPRs from neighbor table, and finally construct hello messages and sent out them acordingly.

#### 2.4.4 Recver Hello Thread

This thread is responsible to receive hello messages, pass the new hello message to Neighbor Table class and MPR selectors Table class, so the neighbor table and MPR selectors table can update according to newly received hello messages.

### 2.5 Vehicle Mobility

A vehicle manager is implemented as a seperate process to manage the locations of all vehicles. Each vehicle node has an agent to communicate with vehicle manage in order to know whether a node is in communication range physically. By enable each node to know the geographical information, we are able to simulate the fact that vehicle are moving within the map and each node is only connected to a number of other nodes from the perspective of communication range.

Some of assumptions that we rely on, and several models that we are using in this project, are listed as follows:

- Wireless links are assumed to be always bi-directional. In other words, any given pare of vehicle nodes are either able to both send and receive message from each other, or cannot communicate at all. There's no omni-directional links simulated in this project.

- Wireless links are simulated by using socket on tux machines, which are either ethernet channel or same-host connections. No wireless media contention is simulated, neither is interference. As a result, the testing data may not truely reflect the using scenario of vehicular network.

- A $5 \times 5$ grid map, as shown in Figure 2, is used to simulate roads, i.e., there are 5 streets and 5 avenues orthogonaly and uniformally aligned on the map. Roads are 120 meters from each other.

- Road ends are headtotail connected. For example, if a vehicle moves to the eastest intersection of an avenue and still goes east, it is relocated to the westest intersection of the same avenue.
- Vehicles can overlap on each other. No collision detection is made and vehicles running on the top of each other simply continues as if there's no accident.

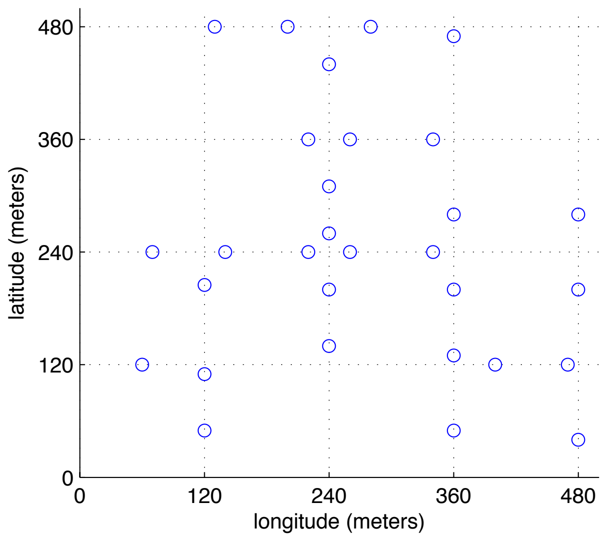


Figure 2: Initial Locations of Vehicles for 30 nodes configuration

### 2.5.1 Data Sharing Between Vehicle Manager and Vehicle Nodes

We put effort in two approaches in order to enable data sharing between vehicle manager and vehicle nodes. The most naive to implement this function is to use a shared file on network file system (NFS) that all the tux hosts running vehicle nodes have access to. However, file I/O has relatively high delay, and when multiply processes are reading from and writing into the same file, there is a probability that some reading operations can fail. Thus we decided to try out a mature cache server Memcached first.

- **Memcached**

  Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects). It is widely used in industry. Web services use it as a cache layer between the web server and database for hot data. It provides APIs for data writing and retrieving.

  We designed to set up a Memcached server as a seperate process for vehicle locations data sharing. A single process, vehicle manager, was to update vehicle locations and update to Memcached server, and each vehicle node has an agent that retrieves locations from the Memcached server. Since Memcached deals with network communication and data ACID feature, the usablitiy and consistence of the data should be very reliable.

  As we use C/C++ to implement everything in our project, we used the C++ client library libmemcached to communicate with Memcached server. We implemented both vehicle manager and vehicle side agent for memcache, and we were able to validate the updating data stored in Memcached server through its concole, except that the retrieving does not work on vehicle side agent. We were not able to make the `Memcache::memcache::get` work. It always complains Timeout error.

  We were not sure whether it was because we were using the API not properly or because there was a bug in libmemcached. After spending some time on it without sucess, we decided to switch to naive way, which is to use shared file on NFS.

- **Shared File**

  The path to the shared file is passed through program argument. The vehicle managers keeps writing into the shared file the newest updated locations for all vehicles at a frequency of 200 milliseconds. The agent in each vehicle nodes keeps reading the shared file and update its own cache. In this way, when the sending threads need to send out messages, they can check on the cache to know which nodes are in communication range. This is pretty simple. We made it work, although, we believe it is not the best design.

### 2.5.2 Vehicle Manager

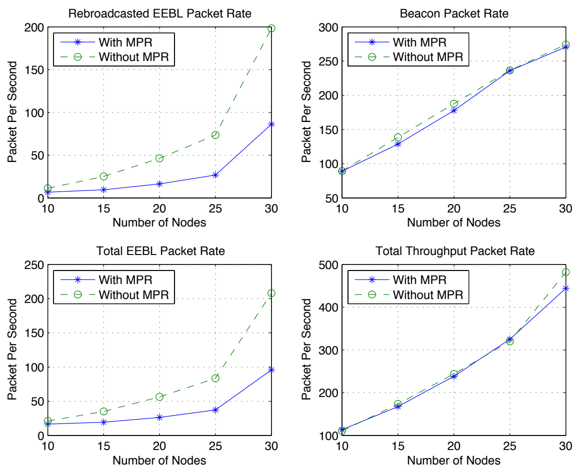
As described in Section 2.5.1, the vehicle manager is in charge of simulating vehicle mobility and keeping the locations in the shared file updated. Before updating the shared file periodically, the vehicle manager needs to compute the new location for each vehicle in the map. At the very beginning, it reads in the initial locations(see Figure 2) of all vehicles from the configuration file. Randomly speed and direction are generated for each vehicle. Each time it updates the shared file, it computes the new location based on the generated speed and the time since last time it updates the shared file. If there is any intersection between new location and old location, a left turn or right turn is possibly generated based on a probability of 0.4. Although randomly generated for each vehicle, the speed itself is mostly constant, except that we considered that the vehicle slows down when it makes turns at intersections. To simplify the model, we simply put the vehicles that decide to make turns pause at the intersection. In other words, if a vehicle needs to turn at an intersection, it stops at the intersection within this round of update no matter how much distance is left from the computation. The direction of that vehicle is then modified and in next update it will be moving.

## 3 Testing Results

To explore how OLSR's MPR concept can impact the EEBL broadcasting in vehicular network, we did some simple tests. We alternated our program into a second version, which does not use MPRs and simply broadcast to all vehicles within communication range. Two versions are compared on 10 - 30 vehicles with mobility. We also tested on a 30 static(without mobility) vehicles configuration in which vehicles form a connected graph.

Figure 3 shows the result in dynamic traffic, in which vehicles are moving with the speed that is normally distributed with the mean of 15 m/s. It looks like the number of rebroadcasted EEBL increases exponentially. The increasing is notable from 25 nodes to 30 nodes. The MPR can greatly reduce the number of EEBL packet rebroadcasted, especially when the number of nodes is as large as 30.

Figure 4 shows the result in static configuration, in which 30 vehicles form a connected graph and do not move. In this way, two versions(with or without MPRs) are in the same situation to be compared. Again, this shows that, with MPR used, the number of EEBL rebroadcasted could be greatly rediced.



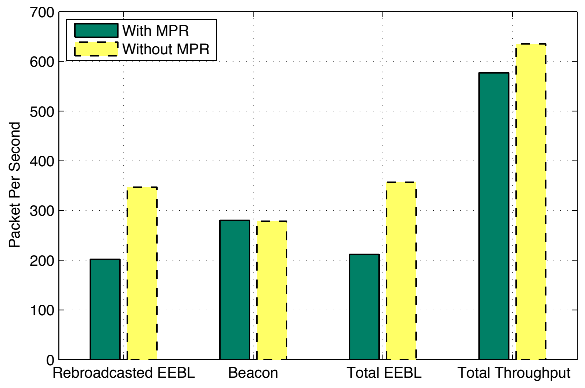
* Total EEBL = EEBL + Rebroadcasted EEBL;

* Total Throughput = Total EEBL + Beacon + Hello Message;

Figure 3: Packet rate for different packets for dynamic traffic

## 4 Conclusion

The test results show that with OLSR's MPR concept used in EEBL rebroadcasting decision, the number of rebroadcasted messages can be greatly reduced. Since in vehicular wireless ad-hoc network, vehicles share the same wireless channel, there is channel contention among vehicles. When vehicles are close to each other, the contention becomes a serious problem and it is important to reduce the number of rebroadcasted messages. From this perspective, using MPR in rebroadcasting in vehicular network can be potentially a good approach.



* Total EEBL = EEBL + Rebroadcasted EEBL;

* Total Throughput = Total EEBL + Beacon + Hello Message;

Figure 4: Packet rate for different packets for static vehicles without mobility

However, on the other hand, as we can see in the 'Total Throughput' part of the plots, the gap between two versions is relatively small compared to the total packet rate. As a result, the ultimate contribution to reducing network load from MPR used in vehicular network is relatively small.

## References

[1] Michele Segata and Renato Lo Cigno. 2011. *Emergency braking: a study of network and application performance.* In Proceedings of the Eighth ACM international workshop on Vehicular inter-networking (VANET '11). ACM, New York, NY, USA, 1-10. DOI=10.1145/2030698.2030700 http://doi.acm.org/10.1145/2030698.2030700

[2] O. Tonguz, N. Wisitpongphan, J. Parikh, F. Bai, P. Mudalige, and V. Sadekar. *On the Broadcast Storm Problem in Ad hoc Wireless Networks.* In Broadband Communications, Networks and Systems, 2006. BROADNETS 2006. 3rd International Conference on, pages 111, Oct. 2006.

[3] Anthony Williams. *Simpler Multithreading in C++0x.* `http://www.devx.com/SpecialReports/Article/38883/1954`

[4] Memcached Community, *Memcached Document.* `http://code.google.com/p/memcached/wiki/NewStart?tm=6`